Attorney Docket No.: 20481-000300US

PATENT APPLICATION

SYSTEM FOR OBFUSCATING COMPUTER CODE UPON DISASSEMBLY

Inventor(s):

Bin Xu
955 La Mesa Terrace, Unit-I
Sunnyvale, CA 94086
U.S. Citizen

Jim Sesma
P.O. Box 2690
White City, Oregon 97503
U.S. Citizen

Robert Freedman
4189 Donald Drive
Palo Alto, CA 94306
U.S Citizen

Weijun Li
687 Ontario Court, #8
Sunnyvale, CA 94087
Citizenship: P.R.China

Assignee:

Preview Systems, Inc.
1195 W. Fremont Avenue
Suite 2001
Sunnyvale, CA 94087

Entity: Small

TOWNSEND and TOWNSEND and CREW LLP
Two Embarcadero Center, 8th Floor
San Francisco, California 94111-3834
Tel: 415-576-0200

SYSTEM FOR OBFUSCATING COMPUTER CODE UPON DISASSEMBLY

COPYRIGHT NOTICE

BACKGROUND OF THE INVENTION

This invention relates in general to computer software and more specifically to a system for preventing accurate disassembly of computer programs.

Computer software manufacturers have a keen interest in protecting their software. Software can be easily copied, in whole or in part, by making digital copies. Other forms of the copying do not require a competitor to copy the actual digital data, but are based on a knowledgeable programmer viewing the instructions within the software to gain information that can allow the programmer to "break" security systems, obtain valuable programming techniques or trade secrets of the software manufacturer, make derivations, manipulate the operation of the original code, etc.

One barrier to copying computer software is that many forms of software are distributed in a format that is not easily decipherable, or readable, by a human.

Figure 1B is an illustration of various forms in the prior art which a computer program, or software, is transformed into during the process of creation, distribution, and ultimate execution of the software on a user's machine.

In Fig. 1B, human readable source code 10 is developed by a programmer who is the original author, and owner, of the work. Such source code is easily readable and understandable by a human programmer since the source code is written in text that resembles plain English with mathematical and logical equations. Many different forms of source code exist today based on many different types of computer languages. "Assembly code" is a form of human-readable code that is closely tied to a specific microprocessor's instruction set. Assembly code has many similarities to source code in

terms of the form translations that the assembly code undergoes prior to being executed. For purposes of this specification, source code and assembly code can be treated similarly, and terminology and concepts associated with source code and assembly code can be interchanged. For example, as discussed below, compilation and assembly are analogous, as are decompilation and disassembly.

Returning to Fig. 1B, source code 10 is compiled by compiler 12. Compiler 12 is a software process that translates human-readable source code to a series of numbers which is, for the most part, unreadable by humans. Source code 10 is thus transformed, or "compiled," by compiler 12 to form the human-unreadable object code. Object code 14 can be linked by linker 20 with other object code modules as illustrated by object code modules 16 and 18 in Fig. 1B. Once the object code modules are linked by linker 20, they form executable program 22. Executable program 22 can be loaded by loader 24 into a user's computer to form executing image 26. Executing image 26 represents the actual numerical information that is executed by a microprocessor within an end-user's computer.

Note that all forms of source code 10 that exists after compilation by compiler 12 are, for the most part, unreadable by a human. In other words, object code modules 14, 16 and 18; executable program 22; and executing image 26 are basically unformatted conglomerations of numbers that are extremely difficult to understand.

However, tools exist to decompile, or disassemble, these unreadable versions of source code. Decompiler 28 can accept the unformatted numbers of object code 14, executable program 22 or executing image 26 and produce a readable version of the original source code program. Such a readable version is referred to as decompiled (or disassembled) code 30. While the decompiled code is usually not as readable as original source code 10, it is a very effective tool for allowing an experienced programmer to understand the operation of the computer program and greatly reduces the amount of time required to copy, hack, or otherwise manipulate source code produced by an original programmer.

Thus, it is desirable to produce an invention which prevents , or reduces the effectiveness of decompilation, or disassembly, of compiled or assembled code.

SUMMARY OF THE INVENTION

The present invention prevents disassembly of computer code. Such prevention includes hiding, masking, or otherwise "obfuscating," the original code. This

helps thwart unwanted parties from making copies of an original author's software, obtaining valuable information from the software for purposes of breaking into the program, stealing secrets, making derivative works, etc. The present invention uses special assembly-language instructions to confuse the disassembler to produce results that are not an accurate representation of the original assembly code. In one embodiment, a method is provided where an interrupt (typically a software interrupt) is used to mask some of the subsequent instructions. The instruction used can be any instruction that causes the disassembler to assume that one or more words subsequent to the instruction, are associated with the instruction. The method, instead, jumps directly to the bytes assumed associated with the instruction and executes those bytes to achieve the original functionality of the program.

A preferred embodiment works with a popular Microsoft "ASM" assembler language and "DASM" disassembler. The instructions used to achieve the obfuscation include software interrupt, "INT," instructions. Using this approach, up to 17 bytes of obfuscation can be achieved with five instructions. Each instruction remains obfuscated until executed and returns to an obfuscated state afterwards.

In one embodiment, the invention provides a method for obfuscating computer program instructions upon disassembly, the method comprising inserting an obfuscating instruction or causing a disassembler to not disassemble one or more bytes subsequent to the obfuscating instruction; and inserting a branch instruction to invoke execution of the one or more bytes subsequent to the obfuscating instruction.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1A illustrates software instructions of the present invention; and

Figure 1B is an illustration of various forms in the prior art into which a computer program, or software, is transformed during the process of creation, distribution, and ultimate execution of the software on a user's machine.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Fig. 1A illustrates software instructions of the present invention.

In Fig. 1A, instructions at 100 illustrate the concept of code obfuscation. Such instructions are included within the body of an assembly language program. A larger portion of the program is illustrated by preceding assembly code 102 and

succeeding assembly code 104. Note that the obfuscating instruction, and associated instructions, can be inserted more than once within the program.

The obfuscating instruction, and associated instructions, include obfuscating instruction 110, jump instruction 112 and hidden code 114. During execution of the assembly code, the assembly program operates as intended by the original programmer until jump instruction 112 is executed. When jump instruction 112 is executed then obfuscating instruction 110 is skipped and execution proceeds at hidden instructions 114. In other words, obfuscating instruction 110 is never executed. Hidden instructions 114 are part of the instructions written by the original programmer and, thus, are part of the original program. Only jump instruction 112 and obfuscating instruction 110 need to be inserted into the original program.

It should be apparent that the program will operate as originally intended with the exception that a few more cycles of processor time are required in order to perform the jump instruction 112. Also, a few more bytes of information are stored in the program every time the technique of the present invention is used to account for jump instruction 112 and obfuscating instruction (or instructions, as described below). The number of hidden instructions at 114 varies with the specific obfuscating instruction, or instructions, employed, as is discussed in detail, below.

Note that jump instruction 112 need not be immediately adjacent to obfuscating instruction 110. Any instruction that directs a processor to obtain the next instruction from within the "hidden" instructions 114 can be sufficient. Also, although the invention is discussed with respect to hidden instructions 114 being immediately adjacent to obfuscating instruction 110, it is possible that obfuscating instructions may act to hide non-adjacent instructions.

The present invention is described with respect to assembly language code in "ASM" format. Such format is produced, for example, by the Microsoft VC++ compiler. It should be apparent that the techniques of the present invention can be adapted for any type of assembler, or source code, or other computer languages and syntax which provide a suitable obfuscation instruction.

By obfuscating code in different places througout the program, it is much more difficult for a programmer to obtain useful information. The decompiler loses synchronization with the instructions and can display missing, or incorrect, instructions in place of the actual ones. With enough portions of the code obscured, a would-be hacker is required to trace through all the code, manually. The debugger (or disassembler) is

expecting the code to return after a jump to a certain instruction, but the code changes the return location causing the debugger to break out of its gui. Two code examples are provided in Table I and Table II:

```
call $+6        ;Highly efficient!
DB OEBh
add dword ptr [esp],6
ret
```

TABLE I

```
call $+12       ;Not efficient.
DB 083h
jmp $+10
DB 08Bh
Inc [esp]
ret
```

TABLE II

A more advanced technique can involve randomly exchanging jump commands in the .ASM file with 'tricky returns.' This requires pushing the destination address instead of altering the esp register like previous examples. This way, this (intelligent) obfuscation macro would not be competing against other macros. By placing the 'tricky returns' where there is already a jump, the byte overhead is reduced.

The instruction "INT 35" has obfuscation properties. Unlike INT 20, no additional data is displayed. In fact, INT's 34-3A or so have the same ability to totally mask three bytes. As an example:

actual code	**the debugger window**
0 JMP 4	0 JMP 4
2 INT 35h	2 INT 35h
4 NOP	7 XOR EAX,EAX
5 NOP	
6 NOP	

7 XOR EAX,EAX

Of course, as much as this is helpful, three bytes of obfuscation is not all that impressive. In tandem with INT 20 though, it is an entirely other story. This example:

jmp $+2
INT 35h
jmp $+2
INT 20h

yielded 14 bytes of obfuscation. Much better! But, then there is this fine example:

jmp $+4
INT 35h
INT 20h

only six bytes long, but yielded an incredible 17 bytes of obfuscation over five instructions. Each instruction remains obfuscated until executed and returns to an obfuscated state afterwards.

Below is some gibberish code that does a fake comparison, then it jumps into the second byte of the compare, which, along with the first byte of the add instruction, cause the program to jump to the byte after the DB. The purpose of this snippet is to confuse the cracker, and in the process obfuscate six bytes. Although unlikely, to avoid collision problems, the me instruction should be switched to jmp.

```
3B EB cmp ebp,ebx
04 00 add al,0h
75 FB jne $-5
83    DB 083h
```

The object of these are only to obfuscate code. They are classified as 'petty obfuscators' because it would be more suitable to reuse a 'great obfuscator.'

To obfuscate four bytes:

```
jmp $+4          ;     Note: this may need byteswapping
DD 0660FBCA3h    ,BSF SP [REG+4bytes]
```

To obfuscate five bytes:

```
jmp $+4
DD 0660FBAA3h    ;BT WORD PTR [REG+4bytes], 1 byte
```

To obfuscate six bytes:

```
jmp $+4
DD 0660FBAA4h    ;BT WORD PTR [REG*4+REG+4bytes],1byte
```

Although the present invention has been discussed with respect to specific embodiments, these embodiments are merely illustrative, and not restrictive, of the invention. The scope of the invention is to be determined solely by the appended claims.